Generating formal specifications for security-critical applications - a model-driven approach

Nina Moebius, Kurt Stenzel, Wolfgang Reif

Angaben zur Veröffentlichung / Publication details:

Moebius, Nina, Kurt Stenzel, and Wolfgang Reif. 2009. "Generating formal specifications for security-critical applications - a model-driven approach." In 2009 ICSE Workshop on Software Engineering for Secure Systems, 19 May 2009, Vancouver, BC, Canada, edited by Bart de Win, Seok-Won Lee, and Mattia Monga, 68–74. Piscataway, NJ: IEEE. https://doi.org/10.1109/iwsess.2009.5068461.

Generating Formal Specifications for Security-Critical Applications - A Model-Driven Approach

Nina Moebius Kurt Stenzel
Wolfgang Reif
Department of Software Engineering and Programming Languages
Institute of Computer Science, University Augsburg
86135 Augsburg, Germany
{moebius, stenzel, reif}@informatik.uni-augsburg.de

Abstract

The SecureMDD approach aims to generate both, a formal specification for verification and executable code, from UML diagrams. The UML models define the static as well as dynamic components of the system under development. This model-driven approach is focused on security-critical applications that are based on cryptographic protocols, esp. Java Card applications. In this paper we describe the generation of the formal specification from the UML model which is then used as input for our interactive verification system KIV. The formal specification is based on abstract state machines and algebraic specifications. It allows to formulate and to prove application-specific security properties.

1. Introduction

Secure cryptographic protocols are very difficult to design [1]. Therefore, formal methods have been proposed as a possibility to prove the security of a protocol mathematically (e.g. [5, 6, 11, 12, 16]). However, often the formal specifications must be written by hand which is error prone, tedious, and requires an expert. Furthermore, the activities of designing, specifying, and verifying a protocol are almost never integrated into a software engineering methodology (exceptions are e.g. [11] [3]).

We propose a model-driven approach, SecureMDD, to design security-critical systems based on cryptographic protocols. The application and the protocols are modeled with UML. From this UML model a formal specification suitable for interactive verification is generated. Furthermore, a runnable Java(Card) implementation of the application can be generated. This paper describes the generation of the formal specification.

The benefits are (1) the possibility to model the application in an object oriented manner with user-defined classes, (2) a formal specification that is tailored to the UML model. A simpler model will result in a simpler specification which makes the formal proofs easier, (3) a formal specification that is an abstract model of the complete application, not only aspects of it, (4) a specification that is suitable for interactive verification and to prove application-dependent security properties, and (5) a software engineering methodology that integrates the verification of security aspects and the generation of executable code. To the best of our knowledge, there exists no other approach that combines all of these benefits.

Section 2 describes the SecureMDD approach in a nutshell, section 3 introduces an example (a copy card), and section 4 (the main part) presents the generation. Section 5 concludes.

2. The SecureMDD Approach

The SecureMDD development method (see Fig. 1) starts with the definition of a platform-independent UML model of the system under development. The modeling is tailored to security-critical applications and defines a UML profile as well as a simple programming language, called Model Extension Language, to describe cryptographic protocols in detail. The modeled application is used to generate code. A model-to model transformation uses the application model as source model and generates two platform-specific models. Since we focus on smart card applications, we generate one platform for the card component as well as one for the terminal. In a next step, we generate JavaCard code (for the card component) as well as Java code (for the terminal) from the platform-specific models (for more details see [14]).

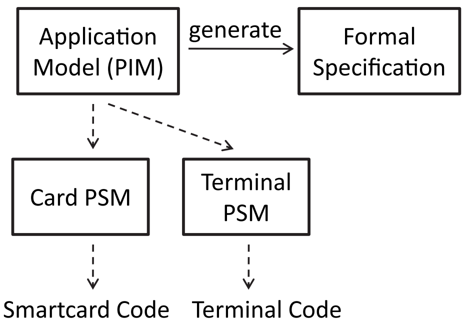

Figure 1. The SecureMDD Approach

In another step, a formal specification is generated automatically from the application model and used for interactive verification with our theorem prover KIV [2]. Technically, the formal specification can be seen as another platform-specific model because the model contains specific details relevant for the verification. All information required to generate a formal specification is already implicitly contained in the application model (e.g. via stereotypes) and no details have to be added later on. The theorems to prove the security of the application have to be added manually. The UML application model only has an informal semantics but a formal semantics is given by the formal specification. The generation of the formal model is implemented with the openArchitectureWare[1] component in Eclipse. One important question is if the generated formal model is correct w.r.t. the generated code. We have some ideas of how to answer this question using refinement. More details about the approach and correctness can be found in [15].

3. Platform-Independent Modeling: Example Copycard

To define the application model we propose to use several UML diagram types to model the different views of the application as well as a UML profile that is tailored to model cryptographic data. The application model gives a view of a system on a platform-independent level.

Figure 4.1 gives an overview of the used diagrams and their interaction. The static part of the application is modeled with class diagrams that import certain data types to model cryptographic data e.g. `Nonce` (for random data) and `Secret` (which hold values that have to be kept secret, for example pass phrases or a pin) and define primitive types such as `Number`. Based on the class diagrams a deployment diagram to describe the structure of the system and the attacker abilities is written. In parallel to the class diagram the cryptographic protocols are developed using sequence diagrams. These diagrams only define the messages exchanged between components of the modeled system. Details of the protocol such as exception handling and processing of a message is not modeled. Activity diagrams refine the sequence diagrams and give a complete description of the dynamic view. A more detailed description of the modeling with UML is given in [15], [13]. We illustrate our approach with an application to make copies, for example at a library. The customer owns a copy card, can load money onto it using a terminal and use his card to pay for copies at the copying machine.

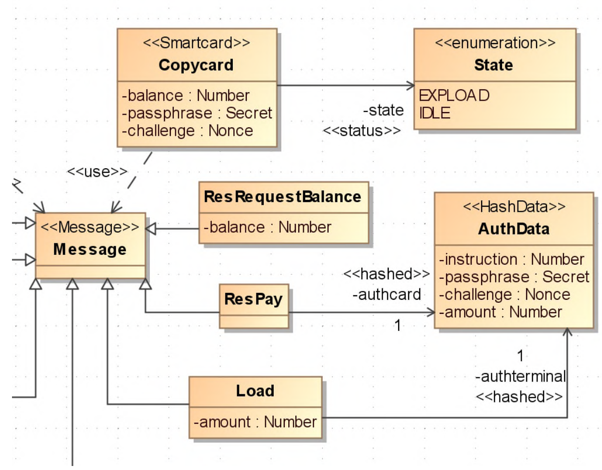

Figure 2. Part of the class diagram of the application model

Figure 2 shows one part of the class diagram for the copy card application. All components of the system are annotated with stereotypes, e.g. the Copycard class has a stereotype ≪Smartcard≫. Message types are defined as classes that are inherited from an abstract class annotated with stereotype ≪Message≫, for example the class `Load`. Classes resp. objects that are going to be hashed during the protocol run are annotated with stereotype ≪HashData≫, e.g. the class `AuthData`. To denote that an object is hashed the stereotype ≪hashed≫ is used. For example, the `Load` message stores a hashed object that was created by hashing an encoding of an object of type `AuthData`.

In Figure 3 a short segment of the activity diagram to load money onto a card is given. For each participating component a swim lane is defined. The segment shows the

[1] http://www.eclipse.org/gmt/oaw/

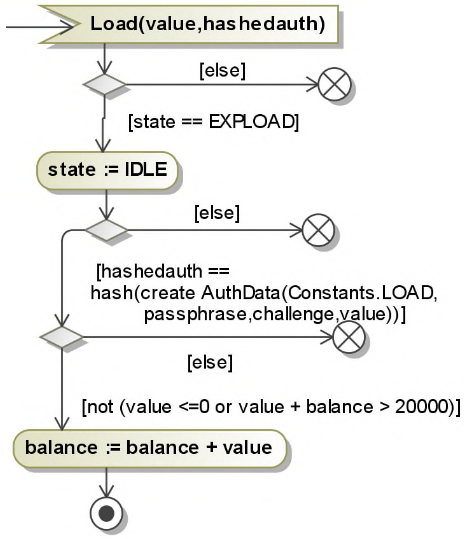

Figure 3. Part of one activity diagram of the application model, swim lane of the copy card

part of the copy card swim lane to process a `Load` message. The protocol is described using the Model Extension Language (MEL). The language allows to e.g. define local variables, assign values to existing fields and local variables, to make checks on the current state of an agent (e.g. the check if the field `state` of the copy card (see Fig. 2)) is set to `EXPLOAD` (see Fig. 3). Furthermore, some predefined operations can be used, for example hashing, encryption and the generation of nonces. MEL has a syntax mixture of UML and Java.

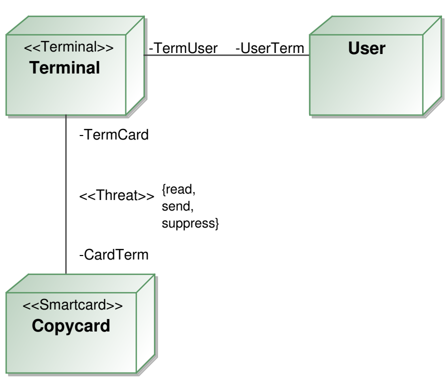

Figure 4. Deployment diagram of the application model

Figure 4 defines the structure of the system and attacker abilities. Each component of the system, i.e. terminal, copy card and user (= the card owner), are given. For each possible communication path a link between the communicating components is defined. Each communication link is annotated with a stereotype ≪Threat≫ that defines if an attacker is able to read messages, suppress messages or send messages over this path. In our example, an attacker has full access on the communication link between copy card and terminal.

4. Generation of the Formal Model

In this section the formal model which is automatically generated from the platform-independent UML model is introduced in detail. In subsection 4.1, an overview of the transformation process is given. In subsection 4.2 the generated communication infrastructure is explained. Subsection 4.3 describes the static part of the formal model, i.e. the data types, components of the application and message types, subsection 4.4 describes the specification of the attacker. In subsection 4.5 the specification of the dynamic aspects of the system are defined using abstract state machines.

4.1. Overview of the Transformation Process

The formal model that is generated from the UML diagrams is an abstract view of the whole UML model (which is a representation of the complete system under development). We aim to formally model the complete system and do not omit parts of the information of the modeled application.

The static aspects of an application, i.e. the components, data types, communication infrastructure and the attacker, are defined using algebraic specifications. The dynamic part of an application, i.e. the cryptographic protocols, are given as an abstract state machine (ASM) [4] (see Fig. 5). The dynamic part can de divided into three parts: an ASM part defining the behaviour of the attacker, an ASM part defining the dynamic behaviour of the remaining components (for example terminal and smartcard), and the trace that is generated by performing ASM steps (for more details see 4.5). Figure 5 shows the components of the formal model and from which part of the UML model they are generated.

The generated formal model is then used to prove the security of the modeled application using our interactive theorem prover KIV [2]. One relevant security property for the copy card example is that the sum of money paid by all customers (and loaded onto the cards) is always greater or equal to the sum of all money that is used for making copies. This implies that it is not possible to load money onto a card without paying the money, for example with a

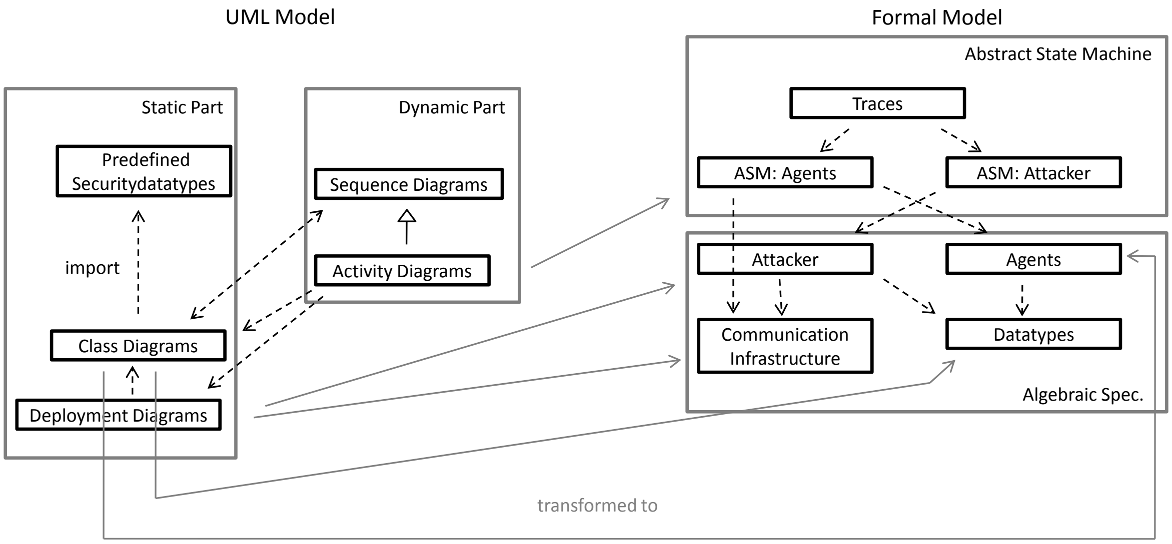

Figure 5. Dependencies between UML Model and Formal Specification

forged terminal. This can be formulated as a theorem over the formal model and can be proved with KIV. We focus on business specific security properties that can not be verified automatically, e.g. using model checkers. To be able to verify these properties it is essential to prove standard security properties (as secrecy or integrity) first.

The formal model automatically generated from the UML models does not use a generic data format but the application-dependent data types defined in the class diagrams. In previous work [9], [8] Haneberg developed an approach to verify the security of an application based on a formal model written by hand. This approach was successfully applied to several case studies but uses a generic data format [9], [8] [10].

4.2. Communication Infrastructure

In the formal model - just like in the real world - only some components of the modeled system communicate with each other. We explicitly model the communication paths between components. Since sometimes it happens that more than one connection between two components exists, we additionally use ports to distinguish the connections. A connection is a pair of two endpoints where an endpoint consists of a component, also called agent, and a port. To each port of an agent an inbox, i.e. a list of messages, is associated that stores the messages that were received in the past but are not yet processed. Sending of a message is modeled by an ASM rule which stores the message in the inbox of the receiver that belongs to the endpoint of the used connection. If an attacker is able to read data sent over this connection the message is also added to his knowledge (see 4.4). All information about connections and ports in the formal model is generated from the deployment diagram.

4.3. The Static Part: Agents and Data types

The agents of the modeled application are defined by a freely generated data type. (listing 1). The information to generate these specifications is taken from the class diagrams. In the copy card example agents of type `smartcard` and `terminal` exist. Furthermore, an agent of type `user` that represents the user interacting with the application is automatically added as well as an attacker agent. As noted earlier we attempt to capture all important aspects of the real world in the formal model. If the application is used in a productive environment a lot of copy cards are issued and several terminals are available to load money onto the cards and pay with it. Thus, an attacker may own several copy cards and use multiple terminals at once to attack our system. If the formal model would assume the existence of only one copy card, we would prove the security for a system that may have security flaws in the real world. For this reason the formal model defines an arbitrary but finite number of cardlets, terminals and users (not shown in the listing). Each instance of an agent is identified by a unique name (given as natural number).

The data of the application, i.e. the messages, predefined security data types, and data types defined in the class diagrams are also translated into algebraic specifications. To reduce the gap between the UML models and the formal model used for interactive verification we use the same data types as in the class diagrams and do not add a generic data

```
agent =
smartcard(. .name : nat)
 with is_smartcard |
terminal(. .name : nat)
 with is_terminal |
user(. .name : nat) with is_user |
attacker with is_attacker;
```

Listing 1. data type to define the components of the modeled application

format like in other approaches, e.g. [16], [9]. This simplifies the work with the formal model.

We only generate formal specifications for those predefined security data types and cryptographic operations that are used in the UML models of the current system under development. For the copy card example, only nonces, secrets and hashing are needed. The predefined security data types do not depend on the concrete application and the transformation is generic for all applications. Listing 2 shows the specification that defines the data type `HashedData` (storing the hashed data) and the `hash` operation. Secrets and nonces are handled in a similar way.

```
HashedData =
mkHashedData(. .hash : HashData);

functions
hash: HashData → HashedData;

axioms
hash(hashdata) = mkHashedData(hashdata);
verifyhash(hasheddata,hashdata) =
(hasheddata = mkHashedData(hashdata));
```

Listing 2. data type to define the data type HashedData and the hashing operation

The data type `HashedData` represents hashed values. As explained earlier, data that can be hashed is of stereotype `HashData` in the UML models. Thus, the constructor of the `HashedData` data type stores a value of type `HashData`. If an agent or the attacker receives a `HashedData` value, he is not able to extract the contained `HashData` value. The functions `hash` and `verifyhash` describe the hashing of a `HashData` value and the check if a `HashedData` value and a `HashData` value correspond.

In the UML class diagram the class `AuthData` is annotated with stereotype `HashData` which means that data of this type is going to be hashed. We define a data type `HashData` that consists of wrappers for all data types marked as `HashData` in the diagrams (listing 3). In this case a wrapper `wrapAuthData` is generated that has a variable `authdata` of type `AuthData` as parameter. The

```
HashData =
wrapAuthData(. .authdata : AuthData)
       with is_AuthData;
AuthData =
mkAuthData(. .instruction : int;
           . .passphrase : Secret;
           . .challende : Nonce;
           . .amount : int)
       with isAuthData;
```

Listing 3. data type to define the data type HashData

wrapping is necessary because the data types marked as `HashData` are usually also used elsewhere, for example as non-cryptographic data types. For this reason each `HashData` type is defined separately. The constructor of the data type has parameters for each field of the type where the UML types of the fields are translated into the concrete types used in the formal model, e.g. fields of type `Number` are translated into fields of type `int`. The order of parameters in the constructor of data types with at most one association to other classes is determined by the order of the attributes in the UML diagram. If more than one association exists the order of the fields have to be given explicitly by defining a constructor in the class diagram. Specifications for non-cryptographical data and the message classes are generated in a similar way.

4.4. The Attacker

One important aspect in our approach is the modeling of the attacker that aims to interfere with the protocol execution and attempts to gain secret information. The attacker is modeled explicitly as a separate agent in the formal model and is associated with a set of data that represents his knowledge (called `attackerdataset`). The `attackerdataset` contains only data that may be relevant to attack the protocol. Non-cryptographic data such as `int`, `string` does not need to be stored explicitly because it can always be guessed by the attacker. For the copy card the `attackerdata` is specified as a data type to store `Secrets`, `Nonces`, `HashData` values as well as values of type `HashedData`.

If the attacker eavesdrops on a communication path and data is sent over that path it is added to the attackers knowledge. To realize this, we generate a function $\ll$ that takes the current `attackerdataset` and the obtained `message`(list) and computes the new knowledge (listing 4, lines 2-5). A `message` is added to the knowledge by computing the union of the old knowledge `adset` and the `analyze` function of the message `msg` and the knowledge `adset` (lines 8-9). If a list of messages is added to the knowledge, this is done recursively for each message (lines 11-12).

```
1  functions
2  . ≪ . : attackerdataset × message
3                → attackerdataset;
4  . ≪ . : attackerdataset
5                × messagelist
6                → attackerdataset;
7
8  axioms
9  adset ≪ msg =
10   adset ⋃ analyze(msg, adset);
11 adset ≪ [] = adset;
12 adset ≪ (msg + msgs) =
13   (adset ⋃ analyze(msg, adset))
14                          ≪ msgs;
```

Listing 4. Function ≪ that adds a message and derived knowledge to the attackerknowledge

The function `analyze` has two input values, one either of type `message` or of type `HashData` and one of type `attackerdataset`(lines 2-5). The `analyze` computes the derived knowledge from a message or hashdata value. Assume that the `message` is of type `Load` (see Sect. 3) and has the arguments `i` of type `int` and `hasheddata` of type `HashedData`. Then, the `hasheddata` value is added to the attackerknowledge. If data of type `AuthData` is analyzed the fields `passphrase` (of type `Secret`) and `challenge` (of type Nonce) are added to the knowledge. The remaining fields are ignored because they are not security-related.

If the modeled application uses encryption, the computation of the new attackerknowledge is more difficult. In this case it may happen that a key (which is contained in a message) allows the decryption of encrypted data that is already stored in the attackerknowledge. The decryption could reveal further keys or other data. Therefore a function is used to derive all data which is accessible with the new key. This example shows one advantage of application-specific generation of a formal model: If possible, the simpler function is generated.

If the attacker is able to send data over a communication path the data to send has to be generated from his knowledge. We generate an application-dependent predicate ≫ which determines whether certain messages can be sent by the attacker. To achieve this the predicate relates all derivable messages to a certain attackerknowledge. A message `Load` containing a value `hasheddata` of type `HashedData` and a value `i` of type `int` (see Fig. 6) is derivable if the `hasheddata` value is derivable (lines 9-10). This is the case if either the `hasheddata` value or

```
1  functions
2  analyze : message × attackerdataset
3              → attackerdataset;
4  analyze : hashdata × attackerdataset
5              → attackerdataset;
6
7  axioms
8  analyze(mkLoad(i,hasheddata),adset)
9    = {hasheddata};
10 analyze(wrapAuthData(authdata),adset)
11   = {authdata.passphrase}
12   ⋃ {authdata.challenge};
13 ...
```

Listing 5. Function `analyze` that computes the set of data that can be derived from a message and the current knowledge

the corresponding non-hashed value (which is stored in the `HashedData` in the formal model) is contained in the attackerknowledge (lines 6-8).

```
1  predicates
2  . ≫ . : attackerdataset
3                × attackerdata;
4
5  axioms
6  adset ≫ hasheddata ↔
7    hasheddata ∈ adset ∨
8    adset ≫ hasheddata.hash;
9  adset ≫ mkLoad(i,hasheddata)
10   ↔ adset ≫ hasheddata;
11 ...
```

Listing 6. Function ≫ that computes which data can be computed from the current attackerknowledge

The information about which communication paths can be read, suppressed by the attacker or used to send data is generated from the deployment diagram. A common approach is to assume that an attacker has full access to all communication links (Dolev-Yao attacker, [7]). Note that we explicitly model the attacker abilities (which is a more realistic model of the real world).

4.5. The Dynamic Part: Abstract State Machine and Traces

The abstract state machine is generated from the activity diagrams and can be divided into the ASM rule describing the attacker and ASM rules for the remaining agents. If an attacker step is executed it is non-deterministically chosen if the attacker sends a message (over a communication path where he is able to send data) or suppresses a message. The message that is sent is generated from the knowledge of the attacker. The suppression of a message means that a non-empty inbox (of a communication path where the attacker is able to suppress messages) is non-determininstically chosen and some messages stored in this inbox are non-determininistically deleted from the inbox.

An ASM step for the copy card consists of the receiving of a message (i.e. taking a message from the inbox), the processing of this message and an update of the state of the copy card agent. An ASM run is a sequence of single steps that generates a trace. A trace is a sequence of states that describe the status of all agents in every step. Thus, a trace describes an arbitrary protocol run that is possible in the real world. The formal model has to be a suitable model that covers all possible traces. Then, if the application is proven to be secure, it is also secure in the real world. To have the same choices (which agent is chosen for the next step or which inbox is used in the current step) in the formal model as in the real world we use a non-deterministical choose-operator. The state of the ASM is the set of states of all agents. The state of an agent is the sum of its attributes and associations, i.e. its fields. The field of an agent is given as dynamic function, i.e. the field `balance` in the UML class diagram is a dynamic function that maps a copy card agent to the value of the field. An update of the field is a modification of the dynamic function. The main ASM rule non-deterministically chooses an agent and calls the ASM rule for this agent. Then, a non-empty inbox of this agent is chosen and the first message is taken. Afterwards, the ASM rule for this message type is called which processes the message and updates the state of the current agent.

5. Conclusion and Future Work

We have presented a model-driven approach to generate a formal specification from a UML model of a security-critical application. The specification can be used to prove security properties with our interactive prover KIV. The generation approach has the advantages that the UML model is much simpler and easier to create than the specification, and that the specification can be tailored to the model which simplifies verification. Future work includes the generation of useful theorems, more tailoring, a comparison with the verification of a hand-written specification, and support for a broader scope of applications. The work is part of our overall SecureMDD approach.

References

[1] R. J. Anderson and R. M. Needham. Programming Satan's Computer. In J. van Leeuwen, editor, *Computer Science Today: Recent Trends and Developments*. Springer LNCS 1000, 1995.

[2] M. Balser, W. Reif, G. Schellhorn, K. Stenzel, and A. Thums. Formal system development with KIV. In T. Maibaum, editor, *Fundamental Approaches to Software Engineering*. Springer LNCS 1783, 2000.

[3] D. Basin, J. Doser, and T. Lodderstedt. Model Driven Security: From UML Models to Access Control Infrastructures. *ACM Transactions on Software Engineering and Methodology*, pages 39–91, 2006.

[4] E. Börger and R. F. Stärk. *Abstract State Machines—A Method for High-Level System Design and Analysis*. Springer-Verlag, 2003.

[5] M. Burrows, M. Abadi, and R. M. Needham. A Logic of Authentication. *Proceedings of the Royal Society of London*, (Series A, 426, 1871), 1989.

[6] David Basin, Sebastian Mödersheim, and Luca Viganò. An On-The-Fly Model-Checker for Security Protocol Analysis. Technical Report 404, ETH Zürich, 2003.

[7] D. Dolev and A. C. Yao. On the security of public key protocols. In *Proc. 22th IEEE Symposium on Foundations of Computer Science*, pages 350–357. IEEE, 1981.

[8] H. Grandy, D. Haneberg, W. Reif, and K. Stenzel. Developing Provably Secure M-Commerce Applications. In G. Müller, editor, *Emerging Trends in Information and Communication Security (ETRICS)*, volume 3995 of *LNCS*, pages 115–129. Springer, 2006.

[9] D. Haneberg, H. Grandy, W. Reif, and G. Schellhorn. Verifying Smart Card Applications: An ASM Approach. In *International Conference on integrated Formal Methods (iFM) 2007*, volume 4591 of *LNCS*. Springer, 2007.

[10] D. Haneberg, G. Schellhorn, H. Grandy, and W. Reif. Verification of Mondex Electronic Purses with KIV: From Transactions to a Security Protocol. *Formal Aspects of Computing*, 20(1), January 2008.

[11] Jan Jürjens. *Secure Systems Development with UML*. Springer, 2005.

[12] G. Lowe. Breaking and Fixing the Needham-Schroeder Public-Key Protocol Using FDR. In *Tools and Algorithms for Construction and Analysis of Systems, Second International Workshop (TACAS)*, pages 147–166. Springer LNCS 1055, 1996.

[13] N. Moebius, D. Haneberg, G. Schellhorn, and W. Reif. A Modeling Framework for the Development of Provably Secure E-Commerce Applications. In *International Conference on Software Engineering Advances (ICSEA) 2007*. IEEE Press, 2007.

[14] N. Moebius, K. Stenzel, H. Grandy, and W. Reif. Model-Driven Code Generation for Secure Smart Card Applications. In *20th Australian Software Engineering Conference*. IEEE Press, 2009.

[15] N. Moebius, K. Stenzel, H. Grandy, and W. Reif. SecureMDD: A Model-Driven Development Method for Secure Smart Card Applications. In *Third International Workshop on Secure Software Engineering, SecSE, at ARES 2009*. IEEE Press, 2009.

[16] L. C. Paulson. The Inductive Approach to Verifying Cryptographic Protocols. *J. Computer Security*, 6, 1998.